LAB MANUAL

for

C Programming and Data Structures

(BT0067)

of the course

BSc IT – Spring 2009

First Semester

Directorate of Distance Education

SIKKIM MANIPAL UNIVERSITY

OF HEALTH, MEDICAL & TECHNOLOGICAL SCIENCES (SMU)

Contents

Software Requirements: (Minimum)

- Windows Operating System (Windows 2000 or Windows XP)
- C or C++ Compiler

Hardware Requirements: (Minimum)

- PIV Computer
- 512 MB RAM
- 40GB Hard Disk

Objectives:

The objective of this manual is to make the student to get hands on experience regarding the application of theoretical concepts learnt in this semester. It also covers the aspects of installation of the respective software, in this case C compiler.

The set of experiments included as part of this lab manual intend to make the students more practical oriented and also to pave the way for using C programming language for real time business applications.

The manual provides the user with the following details regarding the experiments to be completed:

1. Problem Statement
2. Algorithms

The manual also provides the list of commands to be used for editing, typing, and compilation of the C programs given for different compilers.

The time required by the students to complete each and every experiment is also mentioned against the experiment.

The format for preparation and submission of the experiments at the Learning Centers would be as follows and it should be placed as the 'Table of Contents' page in the Record Note Book.

Methodology of Conduct of Practical Exercises

The course titled “C Programming and Data Structures Lab” bearing the subject code BT0067 has 2 credit weight and the practical exercises have to be performed for a duration of 60 Hours. It contains 10 Exercises as given in the Table 1.

Guided and Unguided Exercises

The time allocated for various components is given below. The practical component of “BT0067 C programming and Data Structures Lab” has to be completed within 7 days of 14 sessions; each session has to be for a minimum of 3 hours. You are advised to consult the LC Faculty for the detailed practical schedule.

Type of Activity	**Day(s)**	**Hours**	**Sessions**
General Practice	1	6	2
Guided Exercises	5.5	33	11
Unguided	0.5	3	1
Self Study and Preparation of Record	---	18	--
Total	**7**	**60**	**14**

In the course of these Exercises, the Learning Centre Faculty will provide you guidance, arrange for demonstration (wherever necessary), and clarify your doubts. The course titled “BT0067 C Programming and Data Structures Lab” deals with the theoretical aspects and sample exercises for reference.

You are required to follow the instruction provided by the LC Faculty and observe the demonstration exercises, carry out the exercises, prepare the Practical record book in accordance with the guidelines mentioned and submit the Practical record book to your Learning Centre Faculty. Each Exercise has assessment component, which will be carried out by your LC Faculty. Your attendance is compulsory for these Exercises.

The assessment during the Guided sessions accounts for 70 marks (out of a total of 100 marks for 2 credit Practical subject). In short, during these Guided Exercises, your Learning Centre faculty will offer you guidance and also perform assessment of your work.

The end semester examination will be conducted by the CoE for three hours for 30 marks, which forms the **Unguided Exercise(s) (UGE),** where one or more problem(s) will be assigned to you. The exercise prescribed as unguided could be based on the concepts underlying guided exercises, but need not be exactly the same.

To complete the lab course successfully, you have to score a combined average of 40% (i.e., 40 marks) in both Guided and Unguided parts together and a minimum of 35% in each part (i.e., 25 in marks in Guided part and 11 marks in Unguided part). In case of failure in either or both the parts, you have to redo the concerned part(s).

Table 1

SNo	***Name of the Experiment***	***Date***	***Time***	***Status***	***Remarks***
1	A Simple C Program				
2	Palindrome Number				
3	Matrix Multiplication				
4	Character Search				
5	Calculator Implementation				
6	Stack Implementation				
7	Linked List Implementation				
8	Prim’s Algorithm				
9	Dijkstra’s Algorithm				
10	Heapsort Algorithm				

Faculty Name: **Faculty Signature:**

Lab Administrator Name: **Lab Administrator Signature:**

***External Examiner Name:** ***External Examiner Signature:**

(* To be done at the time of final external examination)

Instructions:

- Each experiment should be performed based on the algorithm outlined for each experiment.
- The faculty or the instructor at the LC should demonstrate the execution process of the software involved.
- The student should be motivated to produce different outputs from the program by providing varying inputs to each experiment and these outputs should be recorded in the individual student report.

All the student records should contain the following details with respect to each and every experiment.

Guidelines:

- **Aim**: It should contain a one or two line description of the problem being solved.
- **Objective**: Should contain the step-by-step procedure or methods followed for completing the experiment.
- **Algorithm**: A pseudo-code should be written presenting the steps involved in solving the problem.
- The detailed executable version of the program should be written by hand and a print out of these executable versions along with sample outputs should be taken; which should be signed by the concerned authority on the same date of experiment and other details should be recorded as per the table mentioned above.
- A record of work should be submitted at the end semester examination.

Ex.No.1: A Simple C Program

Problem Statement:

Write a program that prints the square of an integer.

Example:

Input: n = 10

Output: Square of n = 100

Algorithm:

1. Use an **integer variable n.**
2. Read the value of n from the console using scanf() function.
3. Use the arithmetic operator * (Multiply) to generate the square of n.
4. Use the printf() function to print the output.

Ex.No.2: Palindrome Number

Problem Statement:

Write a program to check whether the given number is a Palindrome.

Example:

Input: n = 101

Output: n is a Palindrome

Input: n = 100

Output: n is not a Palindrome

Algorithm:

A Palindrome number is a number which read from either left to right or right to left gives the same value.

1. Read the number from the console and store it in an integer variable n.
2. Declare two more integer variables mod and n1.
3. Assign the number n to n1.
4. Use While loop
5. Do the modulus operation on n1 and store it in an integer variable mod
6. Do the operation of storing the result of mod operation in step 5 to the variable rev.

7. Continue the steps 5 and 6 until the value of n1 becomes zero.
8. Compare the value of the variable rev to the variable n
9. If the values in both the variables are same, then print the number is Palindrome; else print that it is not a Palindrome.

Ex.No.3: Matrix Multiplication

Problem Statement:

Write a program to multiply two matrices of any order.

Example:

$$\underset{\textbf{Matrix A}}{\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}} \quad \mathbf{X} \quad \underset{\textbf{Matrix B}}{\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}}$$

$$= \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix}$$

Algorithm:

1. Declare three integer arrays
2. Declare three integer variables that would be the indices to the three arrays declared in step 1.
3. Ask the user to enter the dimensions of the two matrices to be multiplied.
4. Now check whether the matrices can be multiplied by applying the following rule:
“Two matrices can be multiplied if the column value of the first matrix is same as the row value of the second matrix”.
5. If the result of step 4 evaluate to true then:

5.1 print a message on the console that the matrices can be multiplied.

5.2 Ask the user to enter the values of the first and second matrices.

5.3 Now initialize all the values of the resultant matrix to zeros.

5.4 Now perform the matrix multiplication on both matrices and store the product in the resultant matrix.

5.5 Print the values of the resultant matrix in a tabular format as shown in the example.

6. If the result of step 4 evaluates to false, then print a message on the console stating that these matrices cannot be multiplied.

7. Exit from the program.

Ex.No.4: Character Search

Problem Statement:

Write a program that will accept a string and character to search. The program will call a function, which will search for the occurrence position of the character in the string and return its position. Function should return –1 if the character is not found in the input string.

Algorithm:

1. Read the input string from the user. This could be done in the following ways:
   - Read an array of characters into a pre-declared character array.
   - Read it using a scanf function
2. Read a character from the user which has to be searched in the above string.
3. Declare an index into the character string or array and use the comparison operator and do the search.
4. If the character is found, update its occurrence in a counter variable, and proceed with the search process until the end of the string is encountered and update the counter variable accordingly.
5. If the character is not found, then print an error message stating that the search character is not found.

Ex.No.5: Calculator Implementation

Problem Statement: Write a program to implement a calculator to perform all the four basic arithmetic operations on integers.

Algorithm:

1. Declare 4 functions add, sub, mul, div that return the result as an integer value.
2. Write the coding in those functions to perform the intended operations.
3. All these four functions should be declared outside the main functions in a single program file.
4. use a switch case function to take the input form the user and perform the appropriate call to the function.
5. Now display the result.

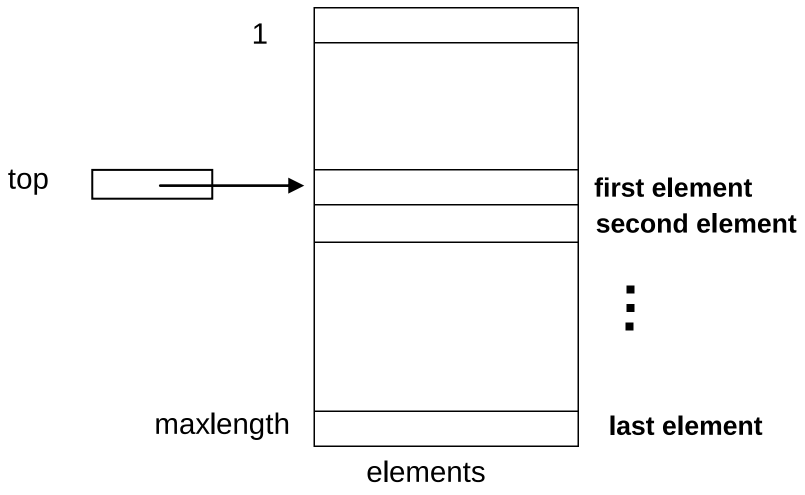
Ex.No.6: Stack Implementation

Problem Statement:

Write a program to implement the Stacks using the concept of arrays.

Algorithm:

A better arrangement for using an array takes account of the fact that insertions and deletions occur only at the top. We can anchor the bottom of the stack at the bottom (high-indexed end) of the array, and let the stack grow towards the top (low-indexed end) of the array. A cursor called *top* indicates the current position of the first stack element.



The following data structure may be used for this implementation:

```
type
STACK = record
      top: integer;
      elements: array[1..maxlength] of elementtype
end;
```

The five basic operations which are to be implemented on a Stack are listed below:

```
procedure MAKENULL(var S: STACK)
begin
     S.top = maxlength + 1;
end; { MAKENULL}

function EMPTY(S: STACK) : boolean;
begin
      if S.top > maxlength then
         return true;
      else
        return false;
end; {EMPTY}

function TOP (var S:Stack) :elementtype;
begin
       if EMPTY(S) then
       error('Stack is empty')
       else
return (S.elements[S.top])
end;  {TOP}

procedure POP( var S: STACK);
begin
       if EMPTY(S) then
       error('Stack is empty')
       else
       S.top = S.top + 1
end; {POP}

procedure Push( x: elementtype; var S: STACK);
begin
       if S.top = 1 then
        error('Stack is full')
    else begin
      S.top = S.top – 1;
      S.elements[S.top] = x
   end
end; {PUSH}
```

Ex.No.7: Linked List Implementation

Problem Statement:

Write a program to implement linked lists using pointers.

Algorithm:

This implementation uses pointers to link successive list elements. This implementation frees us from using contiguous memory for storing a list. However, the price paid is the extra space for pointers.

In this representation, a list is made up of cells, each cell consisting of an element of the list and a pointer to the next cell on the list. If the list is a_1, a_2, …, a_n, the cell holding a_i has a pointer to the cell holding a_{i+1}, for I = 1, 2,…, n-1.
The cell holding a_n has a NULL pointer.
There is a header cell pointing to the cell holding a_1; the header holds no element. In the case of an empty list, the header's pointer is NULL and there are no other cells.

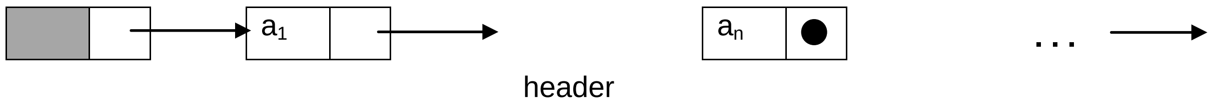


Figure: Pointer implementation of linked lists

The position i will be a pointer to the cell holding the pointer to ai for I = 1, 2, …, n. Position 1 is a pointer to the header, and position END(L) is a pointer the last cell of L (the linked list).

```
type
   celltype = record
    element: elementtype;
    Next:↑ celltype
end;
LIST =↑ celltype
Position =↑celltype;
```

END(L): A function that works by moving the pointer q down the list from the header, until it reaches the end, which is detected by the fact that q points to a cell with a NULL pointer.

```
procedure INSERT(x: elementtype; p:position)
var
    temp:position;
     Begin
          temp = p.next;
          new(p.next);
          p.next.element = x;
          p.next.next = temp
     end;

procedure DELETE(p:position)
 begin
        p.next = p.next.next;
        end; {DELETE}
```

```
function LOCATE(x: elementtype; L:LIST) : position
var
   p: position;
begin
     p = L;
while p.next <> NULL
  if p.next.element = x then
   return(p) {if not found}
  end; {LOCATE}

function MAKENULL(var L: LIST) : position
begin
         new(L);
         L.next = NULL;
         Return(L);
     end: {MAKENULL}
```

Ex.No.8: Prim's Algorithm

Problem Statement:

Write a program to implement the Minimum Spanning Tree Problem using Prim's Algorithm.

Algorithm: An algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

The algorithm continuously increases the size of a tree starting with a single vertex until it spans all the vertices.

- Input: A connected weighted graph with vertices V and edges E.
- Initialize: V_{new} = {x}, where x is an arbitrary node (starting point) from V, E_{new}= {}
- repeat until V_{new}=V:

    - Choose edge (u,v) from E with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, choose arbitrarily)
    - Add v to V_{new}, add (u,v) to E_{new}
- Output: V_{new} and E_{new} describe a minimal spanning tree

Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex *v* in a given graph G =(V, E), defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (*u*, *v*), connecting a vertex v in the set A to the vertex *u* outside of set A. Then vertex *u* is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST.

Choose a node and build a tree from there selecting at every stage the shortest available edge that can extend the tree to an additional node.

MST_PRIM (G, *w*, *v*)

```
1.  Q ← V[G]
2.  for each u in Q do
3.      key [u] ← ∞
4.  key [r] ← 0
5.  π[r] ← NIl
6.  while queue is not empty do
7.      u ← EXTRACT_MIN (Q)
8.      for each v in Adj[u] do
9.          if v is in Q and w(u, v) < key [v]
10.             then π[v] ← w(u, v)
11.         key [v] ← w(u, v)
```

Ex.No.9: Dijkstra's Algorithm

Problem Statement:

Write a program to implement the Single Source Shortest Path problem.

Algorithm

This algorithm is often used in routing. For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).
It should be noted that distance between nodes can also be referred to as weight.

1. Create a distance list, a previous vertex list, a visited list, and a current vertex.
2. All the values in the distance list are set to infinity except the starting vertex which is set to zero.
3. All values in visited list are set to false.
4. All values in the previous vertex list are set to a special value signifying that they are undefined, such as null.
5. Current vertex is set as the starting vertex.
6. Mark the current vertex as visited.
7. Update distance and previous lists based on those vertices which can be immediately reached from the current vertex.
8. Update the current vertex to the unvisited vertex that can be reached by the shortest path from the starting vertex.
9. Repeat (from step 6) until all nodes are visited.

In the following algorithm, the code u := node in *Q* with smallest dist[], searches for the vertex *u* in the vertex set *Q* that has the least *dist[u]* value. That vertex is removed from the set *Q* and returned to the user. dist_between(u, v) calculates the length between the two neighbor-nodes *u* and *v*. *alt* on line 11 is the length of the path from the root node to the neighbor node *v* if it were to go through *u*. If this path is shorter than the current shortest path recorded for *v*, that current path is replaced with this *alt* path. The *previous* array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:           // Initializations
        dist[v] := infinity               // Unknown distance function from source to v
        previous[v] := undefined          // Previous node in optimal path from source
    dist[source] := 0                     // Distance from source to source
    Q := the set of all nodes in Graph    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                 // The main loop
        u := node in Q with smallest dist[]
        remove u from Q
        for each neighbor v of u:         // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)    // be careful in 1st step - dist[u] is infinity yet
            if alt < dist[v]              // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[]
```

Now sequence *S* is the list of vertices constituting one of the shortest paths from *source* to *target*, or the empty sequence if no path exists.

Ex.No.10: Heapsort Algorithm

Problem Statement:

Write a program to implement the Heapsort technique.

Algorithm

Heap sort forces a certain property onto an array which makes it into what is known as a *heap*. The elements of the array can be thought of as lying in a tree structure:
The *children* of a[i] are a[2*i] and a[2*i+1]. The tree structure is purely notional; there are no pointers etc. Note that the array indices run through the "nodes" in breadth-first order, i.e. parent, children, grand-children,....
An array a[i..j] is called a *heap* if the value of each element is greater than or equal to the values of its children, if any. Clearly, if a[1..N] is heap, then a[1] is the largest element of the array.
Now, if a[k+1..N] is a heap, a[k..N] can be made into a heap efficiently:

```
downHeap(int a[], int k, int N)
/*  PRE: a[k+1..N] is a heap */
/* POST:  a[k..N]  is a heap */
 { int newElt, child;
   newElt=a[k];
   while(k <= N/2)   /* k has child(s) */
    { child = 2*k;
      /* pick larger child */
      if(child < N && a[child] < a[child+1])
         child++;
      if(newElt >= a[child]) break;
      /* else */
      a[k] = a[child]; /* move child up */
      k = child;
    }
   a[k] = newElt;
 }/*downHeap*/
```

This operation moves the new element, a[k], *down* the heap, moving larger children up, until the new element can be placed in such a way as to maintain the heap property. The heap can have "height" at most $\log_2(N)$, so the operation takes at most O(log(N)) time.

Heap Sort

This now leads to a version of heap sort known as (bottom-up) heap sort:

```
heapSort(int a[], int N)
/* sort a[1..N],  N.B. 1 to N */
 { int i, temp;
   for(i=N/2; i >= 1; i--)
      downHeap(a, i, N);
   /* a[1..N] is now a heap */

   for(i=N; i >  1; i--)
    { temp = a[i];
      a[i]=a[1]; /* largest of a[1..i] */
      a[1]=temp;

      downHeap(a,1,i-1); /* restore a[1..i-1] heap */
    }
 }/*heapSort*/
```